Chapter 4 Analysis and Design of OO Application Framework on Library Systems Domain

A well-planned framework requires painstaking analysis and design that draw out the generic outlook of the intended use of the framework for future object-oriented applications. A framework project will begin with a system development life cycle that is analysed and designed before implementation. A framework does not expect a perfect product for the first development life cycle because future redesign will eventually improve the design after the framework has been tried and tested in different applications of the domain. A good langauge to follow in analysis and design of a framework is to use the Unified Modelling Language (UML) standard which can better analyse and design object-oriented software development requirements. The first step to take is to include a "*Use Case*" diagram that shows a broad overview of the entire framework domain system. One can determine who or what are the actors dealing with the system and what are their main activities.

4.1 Domain Analysis on Library Systems Domain Model

Domain analysis is the identification and exploitation of commonality across related systems. Domain analysis also includes the process of creating a set of reusable components to be used in the development of systems in the domain (McClure, 1995). The library domain has activities of borrowing a title or reserving a title with borrower's data in, for example, a public library, a school library, a university library or a rental business. These activities are common across related systems and can be implemented as framework components.

Domain analysis is considered a higher-level form of system analysis (McClure, 1995). It is performed for a family of related systems, rather than for one system as is the case for system analysis. In a library systems domain application framework, there exists several subsystems that interact because of communication associations that involve a transaction. Only common abstraction of analysis details should be studied to build components with good reuse potential. Eventually, all the reusable components are interfaced to produce a generic framework for systems in the problem domain. Reusable software components are kept in a library and maintained for future reuse. The domain analysis is carried out to produce reusable software components and should justify itself as an opportunity to include business goals and good system planning to cut cost.

A framework enables the reuse of analysis, design patterns and code. Additionally, a framework includes the most commonly used domain objects (components and classes) with basic associations, such as, relationships between domain classes in a Loan subsystem like Borrower having a Loan on a Copy of a Title. The generic features of a loan subsystem or a reservation subsystem are reusable when software subsystems and components need to interface with each other.

Use case modelling of UML is used to capture reusable requirements of the library systems. A use case model is simple to communicate with domain experts who may not be software architects. Basically, a use case is a scenario that begins with a user of the system initiating some transaction and sequential events. This kind of high-level key abstraction diagram provides an overview to capture the system environment before we define the architecture designs with packages and class diagrams that have unique operating mechanisms. The client-server/dependency, association and composition relationships are mechanisms that display a cooperating structure of classes. Each scenario of the use case can be defined in greater details at the class level.

Our domain analysis should incorporate the library business model with a system architecture that considers future plans of reusability, enhancement, perfective maintenance and integration. Enhancing and maintaining a framework with newly discovered patterns mechanisms, possibly suggested by application developers, requires framework developers to carry out perfective maintenance to the analysis and design model of the framework architecture.

Legacy systems, such as a mainframe database, should be considered as part of the analysis to integrate into the object-oriented library systems domain framework. Day to day activity of the present library system cannot be affected unduly. A software reengineering effort to improve and enhance the domain system is a reason for preventive maintenance and evolution to the current layered system architecture. Due to unforeseen changes in library system functions and rules, a resilient object-oriented architecture can offer flexible changes to a library system and prevent it from going inefficient or obsolete.

4.1.1 Library Systems Requirements with Use Case Modelling

The main purpose of a use case model is to define what a system should do and also allows software developers to communicate effectively with customers. This simple modelling technique is convenient and easily understood without being overwhelmed with too much technical details. Use case modelling is the first step to take in object modelling activities and will start off the software development project.

The actors that actively take part and use the application system are important part of the software environment. Different actors have different roles and different sets of conditions imposed on them. These roles are the impetus for different set of conditions that the software developers should think of incorporating into the eventual system. The roles are, for example, registered library members, visitors or library staff. Basically, actors are borrowers who interact with the library system and whose attributes are kept as state data from the activities of a use case. The behaviour of the system is represented by the use cases which are developed according to the activities, such as loan, reservation and fines participated by the borrowers. Activities that occur in the system should be captured and served on the actors.

In a framework, the roles modelled are generic as frameworks are supposed to serve many types of different roles that might exist in related environments in a developed system domain. For example, a borrowing system would have borrowers, no matter what their designations or roles are in that environment.

The first step to take when analysing a problem domain is to study the system interactions between the users and cases to be solved. UML's use case modelling is the first adopted diagramming technique. This is a very abstract form that is easily understood and communication between developers and end-users is easy.

The framework architect must analyse the problem domain with the domain experts and come up with useful use cases and actors. Otherwise, further development would be futile without getting to know the domain or environment.

One noticeable feature of a framework is having a skeleton business logic itself. It is important especially in software engineering of a sizeable project to have the business logic incorporated into the framework.

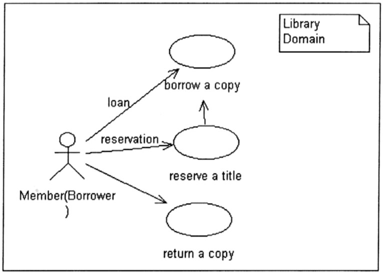

Figure 4.1: Use Case Model of Library System Requirements

In Figure 4.1, a use case model of a general library system reflects the transactions that are carried out based on the requirements of a library member. These abstractions can bring out the salient features of the eventual library system to the early phase of analysis. Every use case of the diagram reflects a system requirement and this mainly includes, for example, to borrow a Copy, to reserve a Title, and to return a Copy which has been loaned.

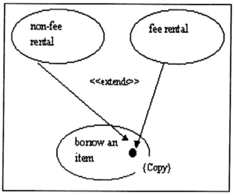

Figure 4.2: Use Case Model of Library Requirement Variation Point

According to Figure 4.2, a use case that depicts a system requirement can produce variable features to an item on loan. The variability is depicted on a use case and is known as a variation point which is represented by a {Copy} tag or a solid dot. The diagram illustrates the application system to have a variation point on the use case represented by two variants, where one variant feature imposes a fee rental for borrowing an item and another variant with a non-fee rental feature. The variability is because of different entity type referenced. The {Copy} variation point tag is referenced by entity types, such as book, CD, journal and etc.

A variation point would likely be implemented as a generic data type that can be parameterised such as C++ <template> or casting Java's base class Object. An abstract type is normally assumed until eventually it is type-defined in the implementation stage or run-time. Based on the type of item or Copy, the decision to impose a fee will be determined dynamically. The <<extends>> stereotype represents extension of variations inserted into the abstract use case component which behaves as a kind of template.

The business rules and variability constraints of the library application system developed from the library systems domain components affects the use case variation point, such as deciding which type of item to impose a rental fee.

4.1.2 Library Systems Architecture with Analysis Modelling

A high-level analysis of the system architecture will provide us with an analysis model. The analysis model consists of analysis types, objects and subsystems together with their relationships defining the architectural structure of the system. At this stage of framework development, we will concentrate on the abstract details of our library analysis objects such as borrowers and items borrowed or reserved.

Once the general requirements have been analysed, agreed and decided upon by the framework engineers and end-users, framework developers can proceed to the second stage of the methodology which is to design the different component packages fulfilling the requirements. Related components would be grouped together into a package to show relevance and most likely scenarios showing close relationship or coupling among the classes. Packaging the relevant components together will assist developers in looking for relevant classes to be reused. The needed package can be quickly passed on to the application developer to be reused because it makes things easier to understand from the package identification.

4.1.2.1 Superordinate System Model

Next, we ask ourselves, how do we relate a use case to subsystems, packages and components? A superordinate system has models to represent the static and dynamic aspects of the layered system as a whole.

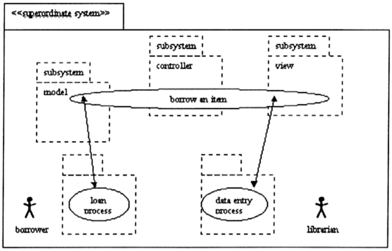

Figure 4.3: Superordinate Use Case in the Superordinate System.

As shown in Figure 4.3, the first included model is a superordinate use case. The use case is traced into one or more superordinate subsystems, for example, borrowing an item from a library is mapped onto different packaged components for a Loan subsystem and a user interface subsystem. We can trace the high-level analysis model to a high-level design model. A use case can span across different layers of the system architecture because we would place the business component like the Loan component of the model subsystem in the business-specific layer, and the application component like the user interface component of the view subsystem in the application systems layer.

By having a clear boundary between subsystems, an interface formed on this boundary subsystem can later help us draw out the communication associations. All use cases should be analysed into a subsystem with defined boundaries, interfaces and controls that help the design and development effort.

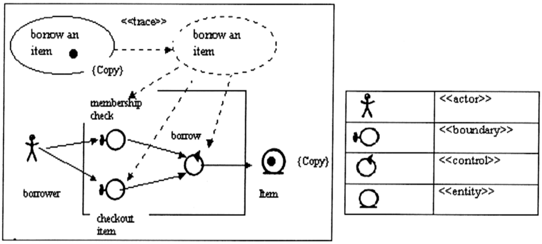

Figure 4.4: Traceability of a Use Case to an Analysis Model Component in a Loan Subsystem.

As shown in Figure 4.4, we trace a use case from the superordinate model to an analysis model for a Loan subsystem component which is comprised of borrowing an item. The actor stereotype, <<actor>>, represents a Borrower. A Loan subsystem activity depicts an interested Borrower whose details are entered into the library system's user interface to check the borrower's membership. The interface is represented as a boundary stereotype, <<boundary>>, and which separates the subsystem from other component system. Internally, the boundary type, possibly a facade, will invoke a set of control logic to process the passing parameters and save the Borrower's entity details into the database as a Loan composite type that the librarian can relate to a Copy as a checkout item. The control stereotype,

<<control>>, depicts the controlling functions of a borrowing activity which serves between the library systems domain model entity types and user interface types.

An entity type is a generic type with the stereotype <<entity>>, reused in many use cases, often with persistent characteristics. An entity type defines a set of entity objects (Jacobson, 1997). The item modelled as an entity has a Copy type and is marked with a variation point which represents a non-fee or fee rental item.

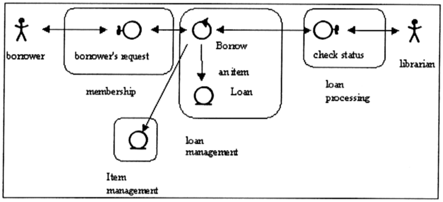

Figure 4.5: Superordinate Analysis Types Mapped into Several Subsystems.

As shown in Figure 4.5, the Loan subsystem is further divided into several subsystems such as library membership, item management, loan management and loan processing. The Loan subsystem is analysed further into separate subsystems for each entity type, namely, Borrower, Item and Loan, which interact within the system. The Membership subsystem is allocated to process a Borrower's request. The Borrower's details are looked up by the Librarian to verify a valid membership and good status, such as, without outstanding loans, fines or membership fee. Hence, this subsystem ensures that he or she is allowed to borrow an item from the library.

The control type, Borrow, acts on a Copy Item and goes through a Loan processing subsystem before it is transformed into a Loan entity type within the Loan Management subsystem. The Item Management is a subsystem that persists the Loan entity into a data storage.

Having these three distinct stereotypes of entity, boundary and control help to develop a robust structure when identifying and specifying types. Actors and entities make up most of the domain model subsystems. In a general library system, the actor is Borrower, and the entity types are Loan, Reservation, Title and Copy. A control type performs a use case scenario.

4.1.2.2 Superordinate Design Model

Relationships among subsystems in a superordinate model provide a high-level communicates-association design, identified by a <<communicates>> stereotype. The subsystems are modelled as packages. An entity type in a package subsystem will exchange messages with a different entity type of another package subsystem. These packages have higher cohesion, which means higher reusability, than a dependency-association relationship. A dependency-association relationship is identified by a <<import>> stereotype. A change to an entity type in a package subsystem can affect the dependent entity in the other package subsystem. These subsystems have a higher coupling between them.

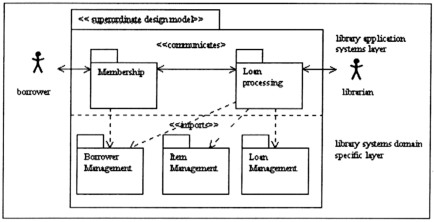

Figure 4.6: Library Systems Superordinate Design Model

Figure 4.6 shows a superordinate design model of the layered system that consists of relationships between package subsystems. The design model is divided into two software architecture layers which are the library application systems layer and library systems domain specific layer, which is regarded as the library component systems with a reusable design model. This high-level architectural design model reflects the static and dynamic aspects of the layered system in terms of package interaction and actor interaction. Each of the application and component systems are part of the package.

Each library application system that is developed can be traced to the related package subsystem. For example, the Membership package subsystem communicates with the Loan processing package subsystem to deal with a loan request activity from an interested Borrower, as depicted in the earlier use case. The Loan request is processed by verification on the Borrower's details and Item's current status, therefore the Loan processing application subsystem will depend on the Borrower Management, Item Management and Loan Management component subsystems to assist the process. Hence, both the Membership and Loan processing package application subsystems will import services and objects from Borrower, Loan and Item package component subsystems in the lower layer. The Loan and Item management package subsystems are placed in the library systems domain specific layer because they contain typical entity types of library business use cases and are commonly reusable in library application systems. The borrower's membership and loan processing package subsystems are placed in the library application systems layer because they offer a coherent set of subsystems that interact according to specific use cases activities in an application system.

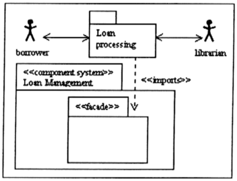

Figure 4.7 : An Application System Import from a Facade.

Figure 4.7 shows a library application system "Loan processing" interacting with the Borrower and Librarian actors and importing from a facade of the component system Loan Management. A facade provides public access or interface to only those parts of the component system, especially from a lower level, which offers reuse of objects and services to higher-level systems.

4.2 System Design on Library Systems Domain Model

A use case in a superordinate model can be decomposed into subordinate use cases for the application and component systems. Each application and component systems may have its own use case model, at least one use case for each system involved. The layers of systems architecture interact together to behave as depicted in the superodinate use case.

Once, the library domain application and component systems have been identified and their facades defined, the superordinate use case is decomposed and defined into subordinate actors and use cases. Therefore, we can begin designing an object model for each system on a high-level sequence diagram.

4.2.1 High-level Scenario of System Design

Object models from a high-level scenario describe how the system is designed and implemented to meet reuser's requirements based on use case and entity stereotypes. System design with an object model consists of abstraction of one or more use cases and entity types that are organised into component subsystems. Each object model defines how responsibilities are allocated to entity types and how relationships will decide the state and attributes of objects. For example, an entity type Loan may have the responsibilities of keeping track on relationships with entities such as Borrower, to know details about when an Item is borrowed, returned or reserved.

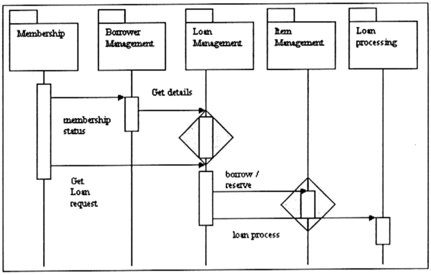

Figure 4.8: Superordinate Design Model Input into a High-level Sequence Diagram.

According to Figure 4.8, we use a sequence diagram to define how a library system superordinate use case can be further decomposed into subordinate use cases. The greyed rhomboids highlight input activities from the superordinate design model when we define the subordinate use cases. This high-level sequence diagram highlights the behaviour of a subsystem interacting with another subsystem. The Membership subsystem interacts with the Borrower actor and the Loan processing subsystem. It also imports from the Loan and Item management packages in the lower layer. Reuse from the component subsystem Loan and Item Management is expressed as an import from a facade.

4.2.2 Static Structure with Class Diagrams

There are three perspectives to modelling a class diagram (Fowler, 1997). Firstly, the conceptual perspective of a class diagram has representations of the concepts in the library systems domain but there is no direct mapping to implementations. The next perspective is the specification of the library systems domain with interfaces that have types instead of classes. A type represents an interface that may have many implementations, because of different library implementation environments. The third perspective is of the implementation of classes. The better approach to take in modelling a class diagram is with taking a system specification perspective.

A class diagram specifies the types of objects in the system and the various kinds of static relationships that exist among them. Once a class diagram has been drawn, the subsequent diagrams with detailed activities will be drawn. If ever the requirement specifications are inaccurately represented at the higher-level analysis and design perspective, the lower level would also be compromised. Inaccurate design of a class diagram will be a waste of effort and would lead to a great deal of maintenance and even redesign. Collaboration effort between framework developers and domain experts is important to avoid any ambiguities on the system requirements.

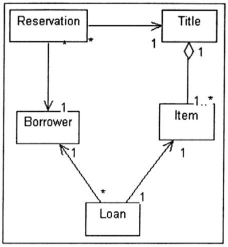

Figure 4.9: Class Diagram for Library Systems Domain

A class diagram must be drawn and tagged with stereotypes from a single and clear perspective. According to Figure 4.9, a class diagram is drawn with the five core library systems domain model object types which are Borrower, Loan, Item, Reservation and Title. These object types with interfaces of the library systems domain have associations between them. Associations represent relationships. For example, a Loan type has a single-valued relationship with a Borrower type, and a Borrower type has a multi-valued relationship with a Loan type. This multiplicity relationship is known as cardinality in UML.

Careful attention should be given to the association between interfaces as they depend on each other to pass object type information, usually this is represented by an arrow as a message passing. According to UML, an association is semantically weak during analysis and should be adorned with more information such as navigability. The uni-directional association is reflected on Loan type which must have a reference to Borrower type.

Aggregation relationship is reflected in the association between Item and Title type. Item type is the server to the client Title which is also known as the aggregate type. The aggregation sign is realised by the diamond symbol on the client type and multiplicity on the server type end of the association. Particularly, the collection of Items is represented this way.

4.2.3 Dynamic Structure with Interaction Diagrams and State Diagrams

Embarking on the design of a framework requires a detailed study of the working details that are shared across systems of the common domain. A useful technique is the 'storyboarding' technique that is modelled with the "interaction diagram" of UML. Interaction diagrams are comprised of two types, "sequence diagram" and "collaboration diagram".

The precise sequence of state activities that occurs in the system is appropriately diagrammed in the "state diagram". Typical activities that happen to an entity and which changes the state of activity is represented in an arrow flow of activities. There are different states that occur in a library system, particularly, the change of state to an Item which is influenced by a Loan or a Reservation by a Borrower. The Reservation scenario can be put into a "state diagram" too as it entails numerous details of activities such as date and waiting period for it. A generic entity type for an application framework development, such as, an Item is a generic representation for a book, magazine, CD, video or anything that can be closely related to such borrowed or loaned item.

For each use case, a collaboration diagram can be created that shows how objects of different types interact . By acknowledging an association between objects occurs, passing of data reference should happen. We can trace between the use case to the analysis type and to the design type. To get more detailed message flow and sequence of action events, the sequence diagram is the solution on the design stage.

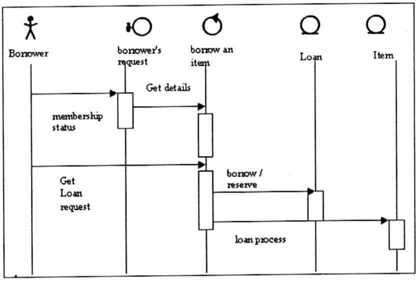

Figure 4.10: Sequence Diagram for a Loan process.

The design model in Figure 4.10 serves to define the entity types from the analysis model into object types which will be implemented later into a more detailed and lower-level sequence diagram. By using a sequence diagram to map an analysis subsystem into a design subsystem, the design types will have a closer match to the intended implementation. With the above diagram, we can expand on the design types with activities that influence a Loan subsystem process.

State diagram is important to a good design model as it portrays the dynamic structure of objects that undergo state transitions, which happens when pre-conditions and post-conditions of objects have to be determined, such as, the date of an object when loaned and when it should be returned.

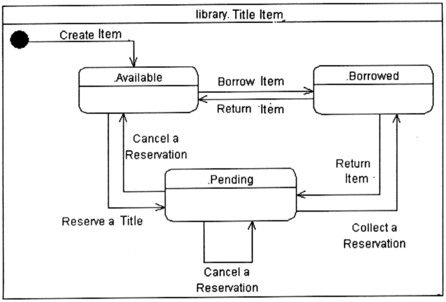

Figure 4.11: State Diagram for an Item.

As shown in Figure 4.11, initially, an Item belonging to a Title is stated as "Available" for loan, and once borrowed, the state of the object is changed to "Borrowed". This state change reflects a pre-condition or post-condition effect on the attribute of the Item. Once returned, the object's state reverts into "Available". Another possibility of "Available" state is the cancellation of reservation due to maturity period of "Pending" state. If the reservation is successfully turned into a loan upon collection by the borrower, the state of the itėm reverts to "Borrowed".

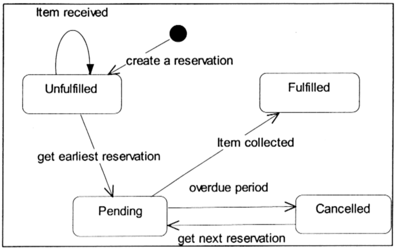

Figure 4.12 : State Diagram for a Title Reservation

Based on Figure 4.12, a Reservation on a Title is accepted from a Borrower and created if all items with the Title are on loan. Initially after reservation, its state is "Unfulfilled" until the Item has been received which can either be a returned loan or a new copy has been purchased. The Item is kept onhold by the library for the earliest Reservation if there is more than one Reservations for the same Title. If the entitled Item on Reservation is uncollected within a given time period and the pending period is overdue, the Item is rendered unfulfilled and becomes available if there is only a Reservation. If there are multiple Reservations, the next earliest Reservation will be retrieved and the Borrower will be informed and the same Item is put onhold with the state "Pending". This rule sequence will continue for the next Reservation until the Item is picked up. All returned Item will trigger an internal mechanism that updates the "Unfulfilled" state of the Title and Item to a "Pending" state.

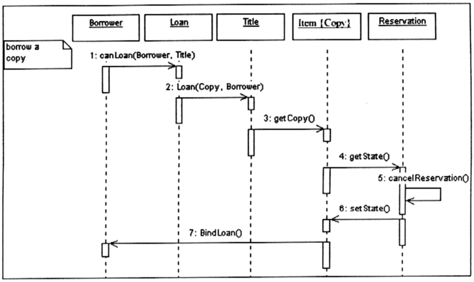

Figure 4.13: Object Interactivity Diagram (Sequence Diagram) for Borrowing an Item.

According to Figure 4.13, the sequence diagram reflects the interactivity of object types that are processed and which will affect the final state of each object. Firstly, a Borrower membership status is determined whether he or she is able to borrow an Item by invoking a method (i.e. canLoan(Borrower, Title)) that references the Borrower's name and the Item's Title as parameters association for message passing.

Subsequently, the Loan's class constructor (Loan(Copy, Borrower)) is called upon to a create a Loan with the Item as a parameter and an association to serve the Borrower. A Loan request will invoke a method (getCopy()) to retrieve the item's details from an aggregate of Items under a Title. Every Item has a unique identification number as key. The state of a Title is determined by invoking a method (getState()) to reference it's state which can be reserved, available or pending. If a Reservation has been made, and the pending period expired, the

Item will revert to available when there is no more reservations. The Item's state can be pending for the next following reservation.

When an Item is borrowed, its state is set to "borrowed" with the method (setState()). The details of a loan is made persistent onto a database with the last method(i.e. bindLoan()). The sequence of methods are invoked in order to process a Loan transaction.

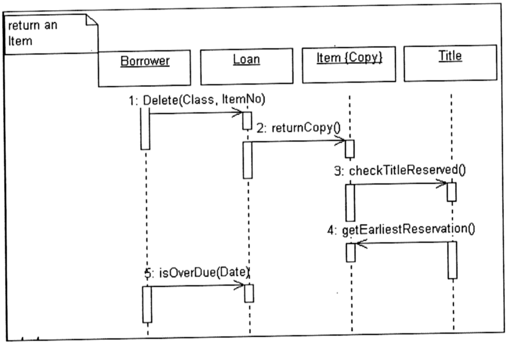

Figure 4.14: Object Interactivity Diagram (Sequence Diagram) for Returning an Item.

Figure 4.14 is a sequence diagram which depicts a Borrower who has returned an Item. Delete is a template class which accepts one of the domain model class type, for example, Loan, Reservation or Borrower. If Loan type is passed to the method, the following method (returnCopy()) of class Loan will be invoked which includes an inner method (checkTitleReserved()) to determine the state of the Title for any reservation. If reservations exist, the earliest reservation will be retrieved by the method (getEarliestReservation()) based on the reservation date created. Lastly, all returned item will be checked by a method (isOverDue(Date)) for late return and imposed fine if it is overdued.

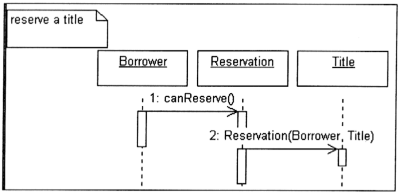

Figure 4.15: Object Interactivity Diagram (Sequence Diagram) for a Title Reservation.

According to Figure 4.15, a sequence diagram is used to model a sequence of two events that takes place when a Borrower places a Reservation on a Title. The first method (canReserve()) will determine whether the interested Title can be reserved. Only when all items of a Title are borrowed would then make it logical for reservation. In order to make a Reservation, the Borrower and the Title types become the message parameters to the Reservation creation (Reservation(Borrower, Title)).

4.3 Summary

The analysis of a library systems domain begins with a use case model for capturing its requirements. A use case is partitioned into different component subsystems in a layered architecture. Component subsystems are analysed to be reusable and still retain relationships among them. The analysis model offers system design at a high level which assists the application framework developer to define the library systems domain layered architecture. A use case is mapped into several packaged components in the analysis model to determine the right stereotypes which can fulfil the individual library systems requirements. With the use of stereotypes, the analysis types such as entity, control and interface will make up the analysis model and represents an abstraction of classes in the system's implementation. High-level design model such as the superordinate model produces diagrams that show relationships between analysis types and mapped them to object diagrams. The various design models of static and dynamic structure of object diagrams define the eventual library systems domain application framework implementation.